Introduction and Background

Welcome to Scala

- *Sc*alable *La*nguage

- A modern language developed for the Java™ platform
  - Interoperates with Java
  - Also .Net, JavaScript versions
- Supports Object Oriented and Functional paradigms
- Many toolkits/frameworks built on Scala
  - Akka
  - Play
  - Slick
  - Spark

A Short History

Pizza

Initial investigation on adding Generics to Java- late 1990s

Generic Java(GJ)

Compiler for superset of Java with Generics and basic FP – 2001

Scala

First experimental release of Scala - 2003

Martin Odersky

Scala 2.0 → Scala 2.11 → Scala 2.12

Mid-2016
Java 8 runtime integration
Ongoing performance improvements

Getting Started

- A first Scala program

A "Singleton object, define class and instance

No requirements on file naming

Hello.scala

```
object HelloWorld {
  def main( args: Array[String] ) {
    println("Hello from scala")
  }
}
```

Semicolon optional as separator at end of line

Type follows identifier in declarations (where type is needed)

Further simplification possible:

```
object HelloWorld extends App {
  println("Hello from scala")
}
```

Running the Program

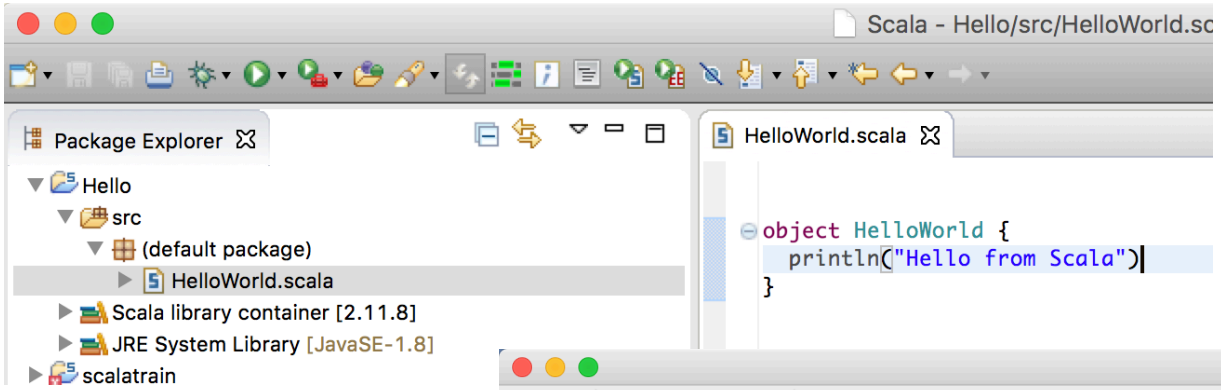
- Standard Scala compile/execution tools available
  - similar to JDK
- Runtime will invoke compiler if suitable class with main method can be found

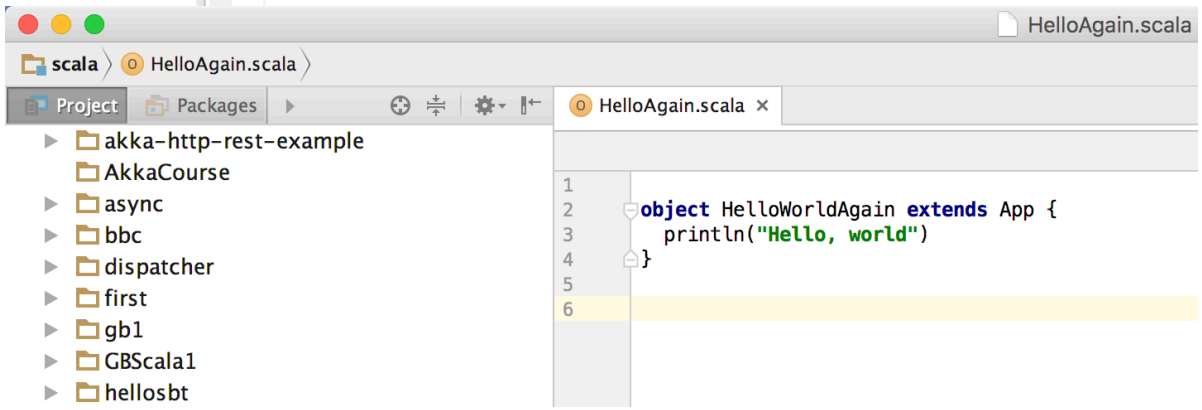
```
$ scalac Hello.scala
$ ls -l
-rw-r--r--   1 george  staff   604 18 Sep 09:27 HelloWorld$.class
-rw-r--r--   1 george  staff   632 18 Sep 09:27 HelloWorld.class
$ scala HelloWorld
Hello from scala
```

```
$ scala Hello.scala
Hello from scala
```

Using an IDE

- Plugins available for common IDEs





The Scala REPL

- An interactive mode for experimenting with Scala

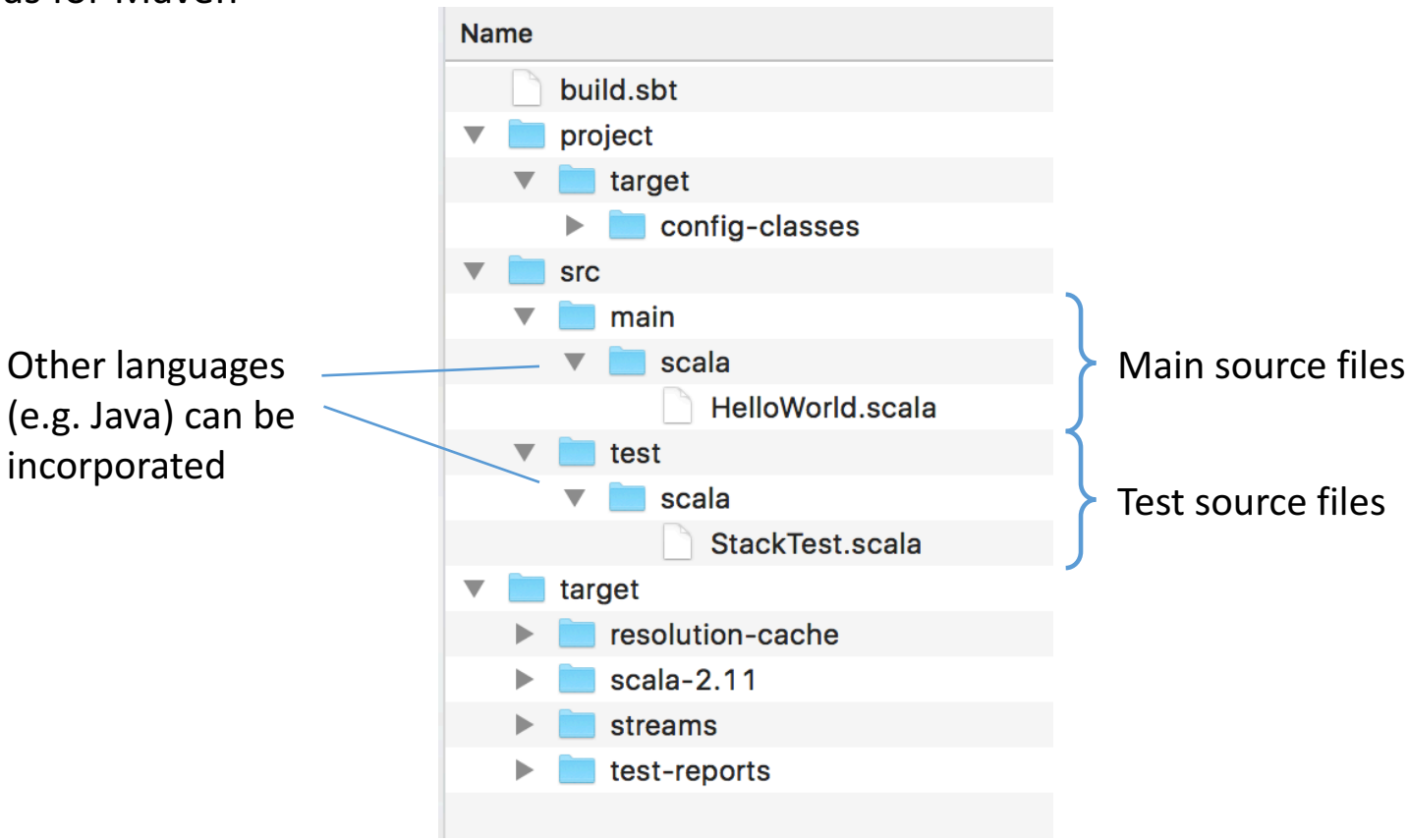
```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_51).
Type in expressions for evaluation. Or try :help.

scala> println("Hello to all...")
Hello to all...

scala> :help
All commands can be abbreviated, e.g., :he instead of :help.
…
:load <path>              interpret lines in a file
:paste [-raw] [path]      enter paste mode or paste a file
:quit                     exit the interpreter
:replay [options]         reset the repl and replay all previous commands
:require <path>           add a jar to the classpath
:type [-v] <expr>         display the type of an expression without evaluating it
…
```

sbt: The Scala Build Tool

- Like Gradle
- Build files use Scala based DSL
  - Unlike Maven which uses XML
- Leverages Ivy for dependencies
- Incremental compilation reduces build times
  - Also server mode of operation
- Basis of Lightbend Activator tool

sbt Project Layout

- sbt has a basic structure for projects
  - Same as for Maven



Working with sbt

```
$ sbt
…
> run
[info] Compiling 1 Scala source to .../target/scala-2.11/classes...
[info] Running HelloWorld
Hello all
[success] Total time: 2 s, completed 08-Aug-2016 15:42:10
>
```

```
…
> test
…
[info] Run completed in 324 milliseconds.
[info] Total number of tests run: 2
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 2, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 2, Failed 0, Errors 0, Passed 2
[success] Total time: 1 s, completed 08-Aug-2016 15:50:41
>
```

The sbt Build File

- build.sbt
  - Written using a Scala DSL

Possible to manage build for several Scala versions

```
name := "Hello World"

version := "1.0"

scalaVersion := "2.11.8"

libraryDependencies += "org.specs2" %% "specs2-core" % "3.8.4" % "test"

libraryDependencies ++= Seq( "org.scalatest" % "scalatest_2.11" % "3.0.0" % "test",
                             "org.scalactic" %% "scalactic" % "3.0.0" )
```

Dependencies can be specified individually or as a list

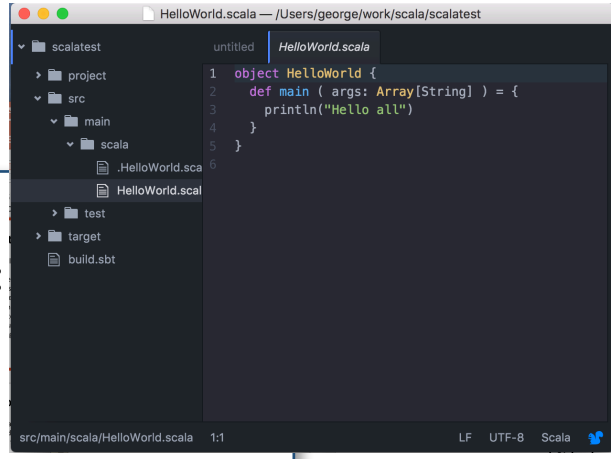
The sbt Console

- Allows Scala REPL interaction with dependencies resolved

```
$ sbt
[info] Set current project to Hello World (in build file:../latest/)
> console
[info] Starting scala interpreter...
[info]
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_51).
Type in expressions for evaluation. Or try :help.

scala> println("Hello there")
Hello there

scala> :quit

[success] Total time: 4 s, completed 08-Aug-2016 16:17:33
>
```

Continuous Mode

- sbt can monitor for changes to files in the build
  - rerun the task if any change is detected
  - prepend ~ to task




```
$ sbt
[info] Set current project to Hello World (in build file:
> ~run
[info] Running HelloWorld
Hello all
[success] Total time: 0 s, completed 08-Aug-2016 16:18:46
1. Waiting for source changes... (press enter to interrupt)
[info] Running HelloWorld
Hello all
[success] Total time: 0 s, completed 08-Aug-2016 16:19:11
2. Waiting for source changes... (press enter to interrupt)

>
```

Basic Language Principles

- Scala is designed to have few principles, but have these principles applied consistently across the language

- Everything is an expression
  - Should yield a value
  - If no value returned, expression is known as a *statement*

- Every value is an object
  - Compiler optimises to use JVM primitive types when appropriate

- Every operation is a method call

Values

- Named pieces of *immutable* storage

```
scala> val myVal = 10
myVal: Int = 10

scala> myVal += 20
<console>:9: error: reassignment to val
              myVal += 20
                    ^

scala> val y: Int = "Hello"
<console>:7: error: type mismatch;
 found   : String("Hello")
 required: Int
       val y: Int = "Hello"
                    ^

scala> val z = { val a = 5; a + 3 }
z: Int = 8
```

Type inferred from initialising expression

val introduces immutable data

Explicitly typed – compiler ensures initialising expression is compatible

Expression may be compound

Variables

- Named pieces of *mutable* storage
  - may be initialised after definition
  - must be initialised before use

- Use of mutable data is discouraged in Scala

```
scala> var myVar: Int = 20
myVar: Int = 20

scala> myVar += 10

scala> myVar
res3: Int = 30

scala> myVar + myVal
res4: Int = 40
```

`var` and `val` may be mixed in expressions

Methods/Functions

- Use def keyword

```
scala> def times2 ( i: Int ) = i * 2
times2: (i: Int)Int

scala> times2 ( 3 )
res17: Int = 6

scala> def upper ( strings: String* ) = strings.map( _.toUpperCase() )
upper: (strings: String*)Seq[java.lang.String]

scala> upper ( "one", "two" )
res18: Seq[java.lang.String] = ArrayBuffer(ONE, TWO)

scala> def sayHello = println( "Hello everyone" )
sayHello: Unit

scala> sayHello
Hello everyone
```

* means variable number of parameters

Unit similar to Java void type

No args so () not required

About Strings

- Scala String type based on java.lang.String
  - Some additional capabilities
- String interpolation allows Scala expressions to be evaluated inside String literals

```
scala> val a = 1
a: Int = 1

scala> val b = 3
b: Int = 3

scala> s"$a + $b = ${a + b}"
res2: String = 1 + 3 = 4

scala> f"$a%2d + $b%2d = ${a + b}%3.2f"
res3: String = " 1 +  3 = 4.00"
```

Simple substitution

Formatted substitution

Regular Expressions

- Powerful notation for working with Strings

```
scala> val s1 = "I like coffee before lunch, and Tea after lunch"
s1: String = I like coffee before lunch, and Tea after lunch

scala> s1.matches( raw".*[Tt]ea.*" )
res14: Boolean = true

scala> s1.replaceAll( raw"[tT]ea", "coffee" )
res16: String = I like coffee before lunch, and coffee after lunch

scala> s1.replaceAll( raw"[tT]ea", "coffee" ).replaceFirst( raw"[Cc]offee", "tea" )
res12: String = I like tea before lunch, and coffee after lunch
```

raw string interpolator prevents expansion of \... sequences in string.

Regular Expressions

- RE Capture groups may be accessed
  - Slightly unusual syntax

```
scala> val s1 = "I like coffee before lunch, and Tea after lunch"
s1: String = I like coffee before lunch, and Tea after lunch

scala> val drink = raw".*([tT]ea|[Cc]offee) before.*([tT]ea|[cC]offee) after.*".r
drink: scala.util.matching.Regex = .*([tT]ea|[Cc]offee) before.*([tT]ea|[cC]offee) after.*

scala> val drink(morningDrink, afternoonDrink) = s1
morningDrink: String = coffee
afternoonDrink: String = Tea
```

Conditional Expressions

- `if` expression
  - Similar syntax to Java/C/C++
  - Similar semantics to ?: operator
  - Yields a value

```
scala> val amount = 25000
amount: Int = 25000

scala> val taxRate = if ( amount < 41000 ) 0.25 else 0.4
taxRate: Double = 0.25

scala> val x = 10
x: Int = 10

scala> if ( x % 2 == 0 ) println("even") else println("odd")
even
```

Unit valued expression/statement

Pattern Matching

- `match` expression
  - similar to switch statement
  - each branch is an expression

```
scala> val month = 5
month: Int = 5

scala> month match {
     |   case 1  => println("January")
     |   case 2  => println("February")
     |   case 3  => println("March")
     |   case 4  => println("April")
     |   case 5  => println("May")
     |   …
     |   case 12 => println("December")
     |   case _  => println("Ooops")
     | }
May
```

No automatic fall-through, so no need for break

Default case represented using _ wildcard

Pattern Matching

- `match` is an operator (method)
  - Part of expression yielding a value

```
scala> def season ( m: Int ) =
     |   m match {
     |     case 1 | 2 | 3    => "Winter"
     |     case 4 | 5 | 6    => "Spring"
     |     case 7 | 8 | 9    => "Summer"
     |     case 10 | 11 | 12 => "Autumn"
     |     case _            => "Weird"
     |   }
season: (m: Int)java.lang.String

scala> season(2)
res5: java.lang.String = Winter
```

Pattern Matching

- Case options can have guards associated
  - Allows continuous ranges of values to be matched

```
scala> def state ( t: Int ) =
     |   t match {
     |     case i if ( i < 0 )              => "ice"
     |     case i if ( i >= 0 && i < 100 ) => "water"
     |     case i if ( i >= 100 )           => "steam"
     |   }
state: (t: Int)java.lang.String

scala> state (120)
res9: java.lang.String = steam

scala> state (-10)
res10: java.lang.String = ice
```

While Loop

- Conventional behaviour
  - Imperative style

```
scala> var j = 1
j: Int = 1

scala> var tot = 0
tot: Int = 0

scala> while ( j <= 5 ) {
     |   tot += j
     |   j += 1
     | }

scala> tot
res10: Int = 15
```

Note var used as variables are mutable

```
scala> var j = 1
j: Int = 1

scala> while ( j <= 5 ) {
     |   if ( j % 2 == 0 )
     |     println(j + ": even")
     |   else
     |     println(j + ": odd")
     |   j += 1
     | }
1: odd
2: even
3: odd
4: even
5: odd
```

Basic for Loop

- Special case of for comprehension
- Body is a statement
  - Evaluated for its side effects
  - Control variable is immutable
- Equivalent to foreach on input Seq

```
scala> for ( a <- 1 to 5 ) println(a)
1
2
3
4
5
```

```
scala> 1 to 5 foreach ( println(_) )
1
2
3
4
5
```